Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №6

Технології розробки програмного забезпечення

ШАБЛОНИ «ADAPTER», «BUILDER», «COMMAND», «CHAIN OF RESPONSIBILITY», «PROTOTYPE»

Виконала:
студент групи ІА-24
Красношапка Р. О.
Перевірив:
Мягкий М. Ю.

Київ 2024

Зміст

Тема: ШАБЛОНИ «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator»

Короткі теоретичні відомості

Abstract Factory (Абстрактна фабрика) – це створювальний шаблон, який надає інтерфейс для створення сімейства взаємопов'язаних об'єктів без вказівки їх конкретних класів. Абстрактна фабрика дозволяє створювати продукти, які належать до певної сім'ї, не залежачи від того, яка саме реалізація буде обрана. Цей шаблон корисний, коли потрібно забезпечити взаємодію різних продуктів, але без залежності від їх конкретних реалізацій.

Factory Method (Фабричний метод) – це створювальний шаблон, який визначає інтерфейс для створення об'єктів, але дозволяє підкласам змінювати тип створюваного об'єкта. Відмінність від шаблону **Abstract Factory** полягає в тому, що **Factory Method** забезпечує створення одиничного продукту, а не всієї сім'ї продуктів. Цей шаблон корисний, коли потрібно делегувати створення об'єкта підкласам і не прив'язуватися до конкретних класів об'єктів.

Memento (Мементо) – це поведінковий шаблон, який дозволяє зберігати та відновлювати попередній стан об'єкта без порушення інкапсуляції. Мементо використовується, коли потрібно зберегти стан об'єкта для подальшого відновлення, наприклад, для реалізації функції "відкату" (undo). Шаблон складається з трьох основних компонентів: **Originator** (об'єкт, чий стан зберігається), **Memento** (об'єкт, що зберігає стан) та **Caretaker** (об'єкт, який зберігає мементо).

Observer (Спостерігач) – це поведінковий шаблон, який визначає залежність типу "один до багатьох" між об'єктами, де зміни в одному об'єкті автоматично сповіщають усіх залежних від нього об'єктів. Це дозволяє створювати події або реакції на зміни, не знаючи точно, які об'єкти повинні реагувати. Шаблон **Observer** часто використовується для реалізації систем подій або підписки.

Decorator (Декоратор) – це структурний шаблон, який дозволяє динамічно додавати нові функціональні можливості об'єктам, не змінюючи їх структуру. Декоратор надає можливість обгорнути об'єкт у новий клас, який додає або змінює його поведінку. Це дає змогу гнучко модифікувати функціональність об'єкта без створення численних підкласів. Шаблон **Decorator** часто застосовується для розширення функціональності елементів у графічних інтерфейсах або для реалізації патернів, що включають додаткові можливості для об'єктів.

Хід роботи

Project Management software (proxy, chain of responsibility, abstract factory, bridge, flyweight, client-server) Програмне забезпечення для управління проектами повинно мати наступні функції: супровід завдань/вимог/проектів, списків команд, поточних завдань, планування за методологіями agile/kanban/rup (включаючи дошку завдань, ітерації тощо), мати можливість прикріплювати вкладені файли до завдань та посилатися на конкретні версії програми, зберігати виконувані файли для кожної версії.

Основні принципи та застосування Abstract Factory (Абстрактна фабрика):

Створення сімейства об'єктів: Шаблон **Abstract Factory** дозволяє створювати родини взаємопов'язаних об'єктів без необхідності вказувати їх конкретні класи. Це забезпечує абстракцію для різних типів об'єктів, які мають спільні характеристики та повинні працювати разом, але їх конкретні реалізації можуть відрізнятися.

Розширюваність: Abstract Factory дає можливість легко додавати нові сімейства об'єктів без зміни існуючого коду. Коли з'являється нова група об'єктів, достатньо створити нову конкретну фабрику, що реалізує абстракційний інтерфейс, і це дозволяє зберегти відкритість для розширення, але захищає від змін у вже існуючій логіці.

Ізоляція клієнта від конкретних класів: Клієнт працює тільки з абстракціями (інтерфейсами), а не з конкретними класами. Це дозволяє зміщувати залежність від конкретних реалізацій, спрощуючи тестування, розширення і підтримку коду, оскільки клієнт не знає, яка саме конкретна реалізація об'єкта використовується.

Конфігурація в залежності від контексту: Замість того, щоб створювати об'єкти вручну, клієнт може використовувати **Abstract Factory** для автоматичного вибору конкретних об'єктів, орієнтуючись на певний контекст або конфігурацію. Це зручно, коли різні об'єкти повинні бути обрані на основі певних параметрів чи умов (наприклад, для підтримки різних платформ чи середовищ).

Застосування в умовах змішування продуктів: Шаблон є корисним, коли потрібно створити програму або систему, яка підтримує декілька варіантів продуктів, які мають спільну функціональність, але відрізняються деталями реалізації. Наприклад, можна створити систему, що підтримує різні варіанти

користувацьких інтерфейсів для різних операційних систем (Windows, macOS, Linux), при цьому зберігаючи загальну структуру.

Реалізація шаблону проєктування для майбутньої системи

Шаблон Abstract Factory (Абстрактна фабрика) є створювальним шаблоном проєктування, який забезпечує інтерфейс для створення сімейств пов'язаних об'єктів, не уточнюючи їх конкретні класи. Він дозволяє створювати різні варіанти продуктів, що належать до однієї категорії, без необхідності зміни коду клієнта, який їх використовує. Абстрактна фабрика визначає загальний інтерфейс для створення об'єктів, тоді як конкретні фабрики відповідають за створення конкретних варіантів цих об'єктів.

Використання шаблону Abstract Factory дозволяє ефективно підтримувати масштабованість системи та гнучкість при додаванні нових варіантів продуктів. Це особливо корисно в тих випадках, коли система повинна підтримувати різні варіанти продуктів або компонентів, але при цьому не повинна залежати від конкретних реалізацій.

У майбутній системі цей шаблон може бути використаний для створення сімейств взаємопов'язаних об'єктів, які належать до різних варіантів платформи, мови програмування чи середовища. Наприклад, в системах, що працюють на різних операційних системах, фабрика може створювати об'єкти, специфічні для кожної ОС, при цьому клієнтський код буде працювати з абстракціями, не залежачи від конкретної реалізації. Це дозволяє забезпечити зручне масштабування та адаптацію системи до нових вимог.

```java
package org.example.projectmanagement.factory;

import org.example.projectmanagement.models.Project;
import org.example.projectmanagement.models.Task;

11 usages  3 implementations  QUIRINO
public interface ProjectFactory {
    3 usages  3 implementations  QUIRINO
    Project createProject(String name, String description, Object... params);

}
```

Рис. 1 – Код інтерфейсу ProjectFactory

```java
package org.example.projectmanagement.factory;

import org.example.projectmanagement.factory.impl.AgileProjectFactory;
import org.example.projectmanagement.factory.impl.KanbanProjectFactory;
import org.example.projectmanagement.factory.impl.RupProjectFactory;

3 usages  QUIRINO
public class ProjectFactoryProvider {

    2 usages  QUIRINO
    public static ProjectFactory getFactory(String methodology) {
        return switch (methodology.toLowerCase()) {
            case "agile" -> new AgileProjectFactory();
            case "kanban" -> new KanbanProjectFactory();
            case "rup" -> new RupProjectFactory();
            default -> throw new IllegalArgumentException("Unknown methodology: " + methodology);
        };
    }
}

```

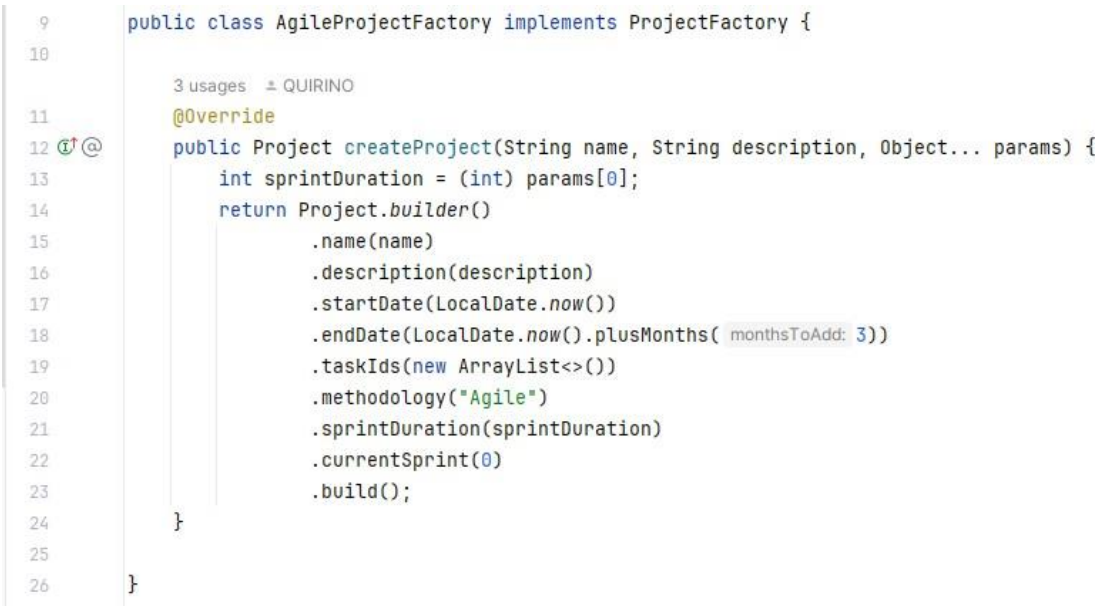
Рис. 2 – Код класу ProjectFactoryProvider

```java
package org.example.projectmanagement.factory.impl;

import org.example.projectmanagement.factory.ProjectFactory;
import org.example.projectmanagement.models.Project;

import java.time.LocalDate;
import java.util.ArrayList;

2 usages  QUIRINO
public class RupProjectFactory implements ProjectFactory {

    3 usages  QUIRINO
    @Override
    public Project createProject(String name, String description, Object... params) {
        String phase = (String) params[0];
        return Project.builder()
                .name(name)
                .description(description)
                .startDate(LocalDate.now())
                .endDate(LocalDate.now().plusYears( yearsToAdd: 1))
                .taskIds(new ArrayList<>())
                .methodology("RUP")
                .phase(phase)
                .build();
    }

}
```

Рис. 3 – Код класу RupProjectFactory

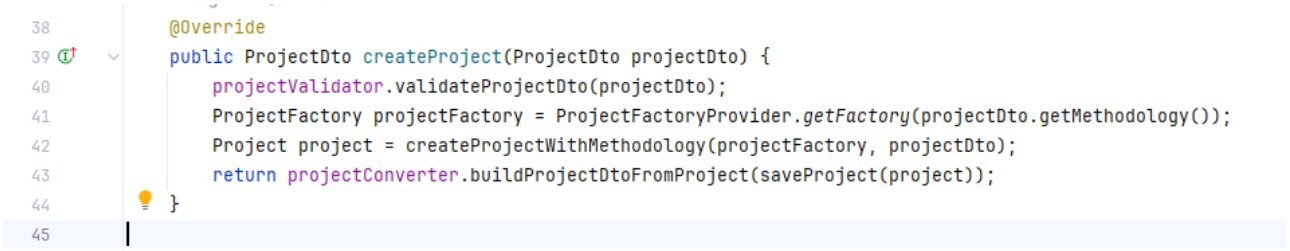
```java
package org.example.projectmanagement.factory.impl;

import org.example.projectmanagement.factory.ProjectFactory;
import org.example.projectmanagement.models.Project;

import java.time.LocalDate;
import java.util.ArrayList;

2 usages  QUIRINO
public class KanbanProjectFactory implements ProjectFactory {

    3 usages  QUIRINO
    @Override
    public Project createProject(String name, String description, Object... params) {
        int workInProgressLimit = (int) params[0];
        return Project.builder()
                .name(name)
                .description(description)
                .startDate(LocalDate.now())
                .taskIds(new ArrayList<>())
                .methodology("Kanban")
                .workInProgressLimit(workInProgressLimit)
                .build();
    }

}
```

Рис. 4 – Код інтерфейсу KanbanProjectFactory



```
9   public class AgileProjectFactory implements ProjectFactory {
10
        3 usages  ± QUIRINO
11      @Override
12      public Project createProject(String name, String description, Object... params) {
13          int sprintDuration = (int) params[0];
14          return Project.builder()
15                  .name(name)
16                  .description(description)
17                  .startDate(LocalDate.now())
18                  .endDate(LocalDate.now().plusMonths( monthsToAdd: 3))
19                  .taskIds(new ArrayList<>())
20                  .methodology("Agile")
21                  .sprintDuration(sprintDuration)
22                  .currentSprint(0)
23                  .build();
24      }
25
26  }
```

Рис. 5 – Код классу AgileProjectFactory



```
38      @Override
39      public ProjectDto createProject(ProjectDto projectDto) {
40          projectValidator.validateProjectDto(projectDto);
41          ProjectFactory projectFactory = ProjectFactoryProvider.getFactory(projectDto.getMethodology());
42          Project project = createProjectWithMethodology(projectFactory, projectDto);
43          return projectConverter.buildProjectDtoFromProject(saveProject(project));
44      }
45
```

Рис. 6 – Приклад використання ProjectFactory

Принцип роботи абстрактної фабрики полягає в наданні інтерфейсу для створення сімейств пов'язаних або залежних об'єктів без вказівки їх конкретних класів. Це дозволяє клієнтському коду працювати з об'єктами через інтерфейси, не знаючи їх конкретних реалізацій. У вашому проекті абстрактна фабрика реалізована через інтерфейс ProjectFactory, який визначає метод для створення проектів. Конкретні фабрики, такі як KanbanProjectFactory, RupProjectFactory та AgileProjectFactory, реалізують цей інтерфейс для створення проектів з відповідними методологіями.

Зображення структури шаблону

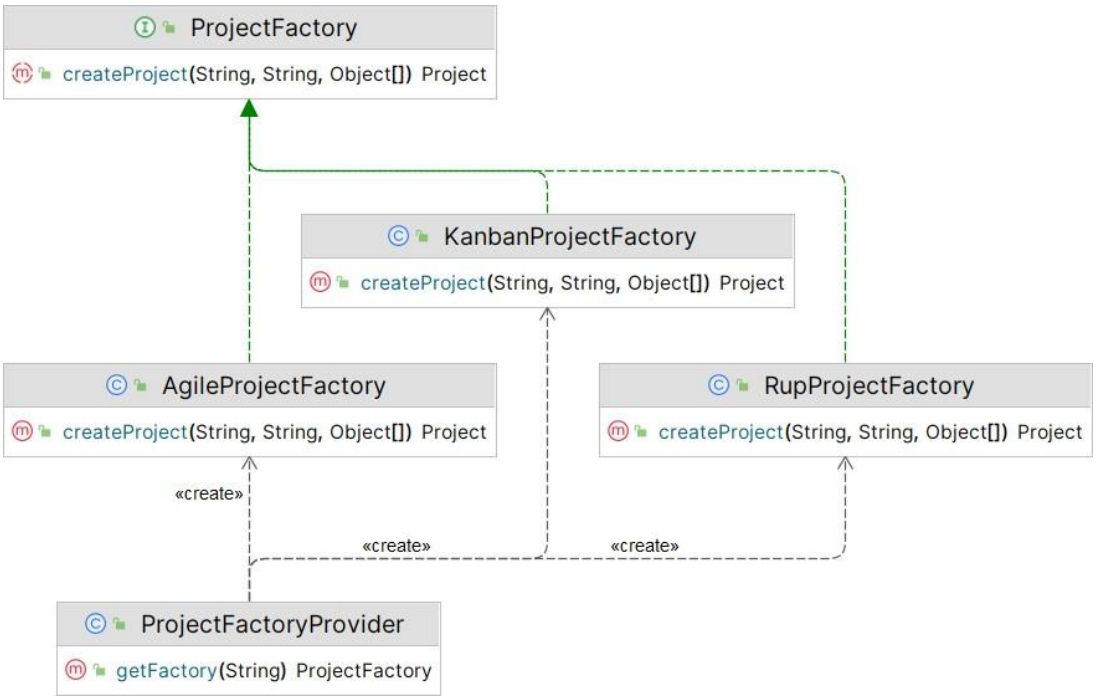


Рис. 7 – Структура шаблону

Посилання на репозиторій:

https://github.com/QUIRINO228/projectManagmentSoftware

Висновок: У результаті реалізації шаблону Abstract Factory було продемонстровано ефективність цього патерна для створення різних типів об'єктів, які належать до однієї сім'ї, залежно від потреб проекту. У конкретному випадку це може бути створення різних типів компонентів інтерфейсу користувача для різних операційних систем або підтримка різних конфігурацій програмного забезпечення.

Кожна конкретна фабрика відповідає за створення певного набору об'єктів, які працюють в рамках своєї платформи. Клієнт взаємодіє лише з абстракцією фабрики, що дозволяє йому отримати потрібні компоненти без необхідності знати, яка конкретно платформа використовується.

Цей підхід забезпечує гнучкість у виборі потрібних реалізацій на основі контексту, дозволяючи спростити додавання нових платформ та компонентів, не змінюючи логіку самого клієнта. Таким чином, шаблон Abstract Factory дозволяє підтримувати чисту та масштабовану архітектуру з мінімальними змінами в існуючому коді при додаванні нових варіантів продуктів чи платформ.